Description of a Molecular Dynamics Simulation System - AA Scale -

Frédéric Boussinot, Bernard Monasse

Description of a Molecular Dynamics Simulation System - AA Scale -

Technical Report

FRÉDÉRIC BOUSSINOT
Mines-ParisTech, Cemef
frederic.boussinot@mines-paristech.fr

BERNARD MONASSE
Mines-ParisTech, Cemef
bernard.monasse@mines-paristech.fr

January 11, 2013

Abstract

A Molecular Dynamics system is presented which is based on Java. A reactive programming framework is used for expressing logical parallelism. The way to define and implement atoms and molecules is described, with some results of simulations showing the stability of the resolution method.

1 Introduction

This report describes a Molecular Dynamics (MD) [5] system which is currently under development at CEMEF[1]. The system has some particularities, unusual with standard MD systems:

- The implementation language is the object-oriented programming language Java [6].
- Parallelism is used at the logical level: complex objects are coded as parallel combinations of more elementary components. For example, molecules are expressed as combinations of atoms, bonds, angles, etc. and atoms are themselves made of several parallel components. This kind of parallelism, that we call *logical parallelism*, is a syntactic means for modular programming, and has to be distinguished from the real parallelism (for example, the one obtained with a multiprocessor computer) which concerns execution. The logical parallelism we use is available through a *reactive programming* (RP) framework [2], based on Java and called *SugarCubes* [8].
- The MD simulations are visualised during execution using the Java3D library [2]. The simulation is actually visualised at each step of the resolution method.

[1]With support from ANR-08-EMER-010 project PARTOUT.
[2]Please note that "reactive" as used here is not at all related to its standard meaning in chemistry.

Rationale for Using Java

A MD system should be interfaced with a 3D visualisation system and with the network (for example, to use clusters of machines), using a communication API (for example, MPI). Moreover, to make the programming task easier, the implementation language should be object-oriented. Our approach requires the possibility to express the behaviour of complex components as parallel combinations of smaller and simpler components. For that purpose, we use a reactive programming approach (see below) and thus we choose a language in which such a RP approach is available. Our choice of Java is motivated by the following reasons: the fact that Java is object-oriented, the existence of Java3D, for 3D visualisation, and the Java-based SugarCubes framework for RP in Java.

The choice of Java is not standard for MD systems, which are usually implemented in FORTRAN (e.g. [1]) or C/C++. Actually, we mainly consider our system as a "proof of concept" system. Provided the existence of a layer for reactive programming in it, we guess that almost any other general-purpose programming language could be used to implement a system similar to ours.

Rationale for Using Reactive Programming

Reactive programming [3] offers a simple framework, with a clear and sound semantics, for expressing logical parallelism. In the RP approach, systems are made of parallel components that share the same *instants*. Instants thus define a *logical clock*, shared by all components. Parallel components synchronise at each end of instant, and thus execute at the same pace. During instants, components can communicate using *instantaneously broadcast events*, which are seen in the same way by all components. There exists several variants of RP, which extend general purpose programming languages (for example, ReactiveC [7] which extends C, and ReactiveML [10] which extends the ML language). Among these reactive frameworks is SugarCubes, which extends Java. In SugarCubes, the parallel operator is very specific: it is totally deterministic, which means that, at each instant, a SugarCubes program has a unique output for each possible input. Actually, in SugarCubes parallelism isimplemented in a sequential way.

The choice of RP, and more specifically of SugarCubes, is motivated by the following reasons:

- MD systems are composed of separate, interacting components (atoms and molecules). It seems natural to consider that these components execute in parallel. In standard approaches, there is generally a "big loop" which considers components in turn (components are placed in an array). This structuration is rather artificial and does not easily support dynamic changes of the system (for example, additions of new components or removals of old ones, things that one can find in modeling chemical reactions).

- In MD simulations, time is discrete, and the resolution method which is at the heart of simulations is based on this discrete time. In RP, time is basically discrete, as it is decomposed in instants. Thus, RP makes the discretisation of time which is at the basis of MD very simple. Note however that resolution steps and instants do not necessarily coincide (actually, with the *Velocity-Verlet* resolution method [11] used in our system, *one* resolution step corresponds to *two* consecutive instants).

- MD is based on classical (Newtonian) physics which is deterministic. The strict determinism of the parallel operator provided by SugarCubes reflects the fundamental determinism of Newtonian physics. At implementation level, it simplifies debugging (a faulty situation can

be simply reproduced). At the physical level, it is mandatory to make simulations reversible in time.

- In classical physics, interactions are instantaneous (this is not the case in Relativity). In RP, interactions are naturally expressed using instantaneously broadcast events. For example, each atom signals its existence to others atoms by generating at each instant an event holding its state. Instantaneity of events means that an event is received in the very same instant it is generated. This is the way instantaneous interactions are coded.

In conclusion, the use of RP for MD simulations is motivated by its following characteristics: modularity of logical parallelism, intrinsic discretisation of time due to instants, strict determinism of the parallel operator, instantaneity of events used to code interactions.

Objectives of the System

The first objective is to design a MD system, with a clear and simple structure, in which users can enter and possibly introduce or change parts. In our system, molecules are built via Java programming (our basic examples are *alkane* carbon-chains). The object-oriented character of Java is essential to simplify the definition of new molecules by programming.

The second objective is to get a system with a minimum of approximations, in reference to the well-established notions that are found in the litterature (for example, the notion of a potential). In this respect, great care is put on the *stability* of the resolution method which we choose to be, as it is generally the case in MD systems, the *Velocity-Verlet* method.

The third, longer-term, objective is to extend the current system to a *multi-scale, multi-time* system, in which molecules at different scales of description - "all-atom" (AA), "united-atom"(UA), and "coarse-grain" (CG) - can be simultaneously simulated. Multi-scale approaches [4] are a way to simulate during long periods of time molecular systems composed of loosely-coupled components (with rare interactions). Our objective is to allow scale changes that preserve fundamental quantities (e.g. energy), putting the focus on the definition of potentials.

In this report, we do not consider anymore multi-scale aspects which are left for future work, and we limit ourselves to the AA scale.

Structure of the Paper

The report is structured as follows: Section 2 to 9 describe the system. Examples are given in Section 10. Finally, Section 11 gives some tracks for future work and concludes the report.

Remarks that can be skipped in a first reading are put between symbols ▶ and ◀.

2 General Structure of the System

The system is implemented as a set of Java classes structured as follows:

- **Constants.** The 3 basic internal physical units of the system are: nanometer (10^{-9} meter), picosecond (10^{-12} second), and dalton (1.6×10^{-27} kilogram). The Java Interface `Constants` contains a set of definitions shared everywhere in the system. The constants of the OPLS force field [9] are used through several functions (e.g. `OPLSBond`) defined in `Utils`. The constants defining the maximum distance to which atoms belonging to the same molecule are not subject

to LJ potential (`maxDistanceAA`) is also defined in `Constants`. The events associated to LJ potential (e.g. `CSignal`) are defined as SugarCubes events.

- **Basic definitions.** The basic definitions used for simulation components are grouped in `Icobj3D`. The physical state of the component is described by several fields: coordinates `x,y,z`, velocity `sx,sy,sz`, and mass `mass`, which are all of the Java `double` numeric type. Three double fields `fx,fy,fz` are defined to collect the forces exerted on the component (il any). The field `behavior` of type `Program` holds the SugarCubes program associated to the component. Fields for Java3D are also present (e.g. `appearance` and `scene`) and are used if the component is drawn on screen. Two events are defined: `killSignal` to kill the component and `constraintSignal` to constrain it. Atoms and molecule components extend `Icobj3D`.

  `Workspace3D` extends the basic class `Applet` of Java and define basic simulations, called *workspaces*. Components of workspaces are held in stack `icobjStack`. The basic Java3D scene, universe, and canvas are fields of the workspace. A workspace also contains an engine (class `Engine3D`) which embeds a SugarCubes reactive machine (`SC.machine`) to execute SugarCubes programs. The typical instruction to make a SugarCubes program `p` run by the machine is to call `getMachine().addProgram(p)`.

  Class `Simulation` extends `Workspace3D` by introducing several means to control simulations; we consider these means as secondary and do not describe them further.

- **Resolution.** Two basic interfaces `Method` and `Equations` specifie the general way to use resolution in the system. Class `Newton` implements Newton's second law in the system. The *Velocity-Verlet* method is coded by class `VerletV`.

- **Atoms.** The general structure of atoms is described by class `Atom`. Classes `C`, `H`, and `O` corresponds to specific atoms (carbon, hydrogen, oxygene). Class `CollectInteractions` describes the way Lennard-Jones interactions between atoms are collected. Class Constraints exerted on atoms (produced by the molecules components, bonds, angles, and dihedrals) are collected using `CollectConstraints`. To use the *Velocity-Verlet* method, an atom must perform action `Compute`.

- **Lennard-Jones.** The Lennard-Jones potentials are defined by interface `Potential` and class `LJPotential`.

- **Bonds, valence angles, and dihedrals.** Bonds are defined through classes `Bond` and `Spring`. Specific harmonic bonds are defined by `HarmonicSpring`. Valence angles are defined by the two classes `Angle` and `HarmonicAngle`. Finally, class `Dihedral` defines dihedrals.

- **Molecules.** General template of molecules is defined in class `Molecule`. Carbon chains are specific molecules, made of a backbone of carbon atoms to which hydrogen atoms are linked. They are called alkanes, and their linear structure is CH3-(CH2)*n-CH3. Construction of alkane is described in class `CarbonChain`. Acid molecules are carbon chains in which the bottom part is changed in a CO2 group of atoms (CH3-(CH2)*n-CO2). They are defined in class `Acid` obtained from `CarbonChain` by just redefining the building of the bottom part.

- **Auxiliary classes.** Class `Units` contains several functions related to physical units. Class `Utils` defines a set of auxiliary functions, among which are the ones to manipulate 3D vectors,

of Java3D class `Vector3d`. Class `MDContext` defines context for simulation: time-step, resolution method, and scale. Appeareance on screen is controlled by `Paint3D`. Rotation related methods are defined in class `Rotation`. Class `Self` is used by atoms to communicate their state to others. Class `Printer` contains function to print various kind of information. Class `Value3` describes triples of `double` values.

The force field used is resumed in Fig. 1. It is based on the OPLS force field [9] , with three changes, in red in the figure. The rationale for these changes is to minimise the energy of the carbon chains built in Sec. 9.1.

▶

	C-C-H	0.1569	109.5	
The genuine OPLS values are:	H-C-H	0.138072	107.8	
	C-C-C-O	-1.336	0.0	0.0

◀

Bond	strength	length	
C-C	112.13119999999999	0.15289999999999998	
C-H	142.256	0.10900000000000001	
C-O	133.888	0.141	
Valence	**strength**	**angle**	
C-C-C	0.2441364	112.7	
C-C-H	0.1569	108.40891312174834	
H-C-H	0.138072	110.51231628706842	
C-C-O	0.20920000000000002	109.50000000000001	
O-C-O	0.3874384	111.5	
Dihedral	**A1**	**A2**	**A3**
C-C-C-C	0.007280160000000001	-6.56888E-4	0.0011673360000000002
C-C-C-H	0.0	0.0	0.001531344
H-C-C-H	0.0	0.0	0.0013305120000000003
C-C-C-O	0.0	0.0	-0.0018632746666666668
H-C-C-O	0.0	0.0	0.001958112
LJ	**ϵ**	**σ**	
ljC-C	2.7614400000000003E-4	0.35	
ljH-H	1.2552E-4	0.25	
ljO-O	7.112800000000001E-4	0.307	
ljC-H	1.86188E-4	0.2958	
ljC-O	4.43504E-4	0.3278	
ljO-H	2.97064E-4	0.277	

Figure 1: OPLS-based AA force field (internal units)

3 Resolution Method

Interface `Method` specifies a unique method `step` which has to be called to perform resolution. Note that the resolution method used in the system (*Velocity-Verlet*) needs two calls of `step` to compute one time-step:

```
public interface Method
{
    public abstract void step ();
}
```

Interface `Equations` specifies two methods to obtain the state and the accelaration of an atom:

```
public interface Equations
{
    public abstract double[] getState ();
    public abstract double[] getAcceleration ();
}
```

The class `Newton` implements Newton's law $F = m\mathbf{a}$, where F is the force applied to an atom, m its mass, and $\mathbf{a}$ its acceleration:

```
public class Newton implements Equations
{
    final Icobj3D icobj;
    final double [] state = new double [6];
    final double [] acceleration = new double [3];
    public Newton (Icobj3D icobj)
        {
            this.icobj = icobj;
        }
    public double [] getState ()
        {
            state[0] = icobj.x;
            state[1] = icobj.sx;
            state[2] = icobj.y;
            state[3] = icobj.sy;
            state[4] = icobj.z;
            state[5] = icobj.sz;
            return state;
        }
    public double [] getAcceleration ()
        {
            double mass = icobj.mass;
            acceleration[0] = icobj.fx / mass;
            acceleration[1] = icobj.fy / mass;
            acceleration[2] = icobj.fz / mass;
            return acceleration;
        }
}
```

► We use Java interfaces because one could imagine to implement other resolution methods. Actually, this has been done for Euler and Runge-Kutta methods, but, for simplicity, we do not consider them here. ◄

3.1 Velocity-Verlet Method

Let $\mathbf{r}$ be the position (depending of the time) of an atom, $\mathbf{v}$ its velocity, and $\mathbf{a}$ its acceleration. The *Velocity-Verlet* resolution method is defined by the following equations, where Δt is a time interval:

$$\mathbf{r}(t + \Delta t) = \mathbf{r}(t) + \mathbf{v}(t)\Delta t + 1/2\mathbf{a}(t)\Delta t^2 \quad (1)$$

$$\mathbf{v}(t+\Delta t) = \mathbf{v}(t) + 1/2(\mathbf{a}(t) + \mathbf{a}(t+\Delta t))\Delta t \quad (2)$$

Implementation is possible in two steps:

1. Compute velocity at half of the time-step, from previous position and acceleration, by: $\mathbf{v}(t + 1/2\Delta t) = \mathbf{v}(t) + 1/2\mathbf{a}(t)\Delta t$. Then, use the result to compute position at full time-step by: $\mathbf{r}(t+\Delta t) = \mathbf{r}(t) + \mathbf{v}(t+1/2\Delta t)\Delta t$.

2. Get acceleration $\mathbf{a}(t+\Delta t)$ from forces applied to the atom, and compute velocity at full time-step using velocity at half time-step by: $\mathbf{v}(t+\Delta t) = \mathbf{v}(t+1/2\Delta t) + 1/2\mathbf{a}(t+\Delta t)\Delta t$

Our implementation of *Velocity-Verlet* uses two instants: first step is performed during the first instant, and second step during the second instant. At the end of the second step, velocity and acceleration are saved for the next step. Acceleration is actually obtained (by Newton's law) from the sum of the forces applied to the atom during the second step only.

The *Velocity-Verlet* resolution method is coded by the class `VerletV` defined as:

```
public class VerletV implements Method, Constants
{                                                                    2
   final Atom atom;
   final Equations eqs;                                              4
   int currentStep = 0;
   double state [] = new double [6];                                 6
   double acceleration [];
   boolean initialised = false;                                      8
   double prevState [] = new double [6];
   double prevAcceleration [] = new double [3];                      10
   public VerletV (Equations eqs,Atom atom)
      {                                                              12
          this.eqs = eqs;
          this.atom = atom;                                          14
      }
   double timeStep ()                                                16
      {
            return atom.molecule.context.timeStep;                   18
      }
    void initialise ()                                               20
      {
            initialised = true;                                      22
            prevState[0] = atom.x;
            prevState[1] = atom.sx;                                  24
            prevState[2] = atom.y;
            prevState[3] = atom.sy;                                  26
            prevState[4] = atom.z;
            prevState[5] = atom.sz;                                  28
            currentStep = 0;
      }                                                              30
    public void step ()
      {                                                              32
            if ( !initialised ) initialise ();
          currentStep++;                                             34
          if ( currentStep == 1 ) step1 ();
          else if ( currentStep == 2 ) {                             36
               step2 ();
               currentStep = 0;                                      38
          }
      }                                                              40
    void step1 ()
       {                                                             42
            double dt = timeStep ();
```

```
            state = eqs.getState ();
            // set intermediate speed sx,sy,sz
            for ( int i = 1, k = 0; i < 6; i += 2, k++ ) {
                state[i] = prevState[i]+0.5*prevAcceleration[k]*dt;
            }
            // set position x,y,z
            for ( int i = 0, k = 0; i < 6; i += 2, k++ ) {
                state[i] = prevState[i] + state[i+1] * dt;
            }
            atom.resetForce ();
        }
    void step2 ()
        {
            double dt = timeStep ();
            acceleration = eqs.getAcceleration ();
            // set speed sx,sy,sz
            for ( int i = 1, k = 0; i < 6; i += 2, k++ ) {
                state[i] = state[i] + 0.5 * acceleration[k] * dt;
            }
            // memorise for next step
            prevState = state;
            prevAcceleration = acceleration;
        }
}
```

The two method `step1` and `step2` corresponds to the two steps of the resolution method. Note that the forces applied to the atom are simply reset at the end of the first step; actually, only the forces collected during the second step (with positions of atoms computed during first step) are used to compute atom velocity.

► The time-step used during resolution is evaluated at each resolution step, by calling the `timeStep` method, and not fixed at construction. This is to allow time-steps to dynamically change, which is a necessity for multi-scale approaches. ◄

4 Atoms

In this section, one first considers the class `Atom` that defines generic atoms. The processing of LJ potentials is not defined for generic atoms. Second, one considers the class `C` of carbon atoms, which extends `Atom` by defining the way LJ potentials are processed. Definitions of hydrogen and oxygen atoms are very similar and thus not described here.

4.1 Generic Atoms

The class `Atom` extends `Icobj3D` and defines generic atoms. It defines a field `molecule` instance of `Molecule` which is the molecule to which the atom belongs. The kill event of the atom is set to the event which kills the molecule. The behavior of the atom is defined as a loop which cyclically collects the constraints sent to the atom (by generating `constraintSignal`), computes the new state of the atom (class `Compute`), and paint the atom on screen (`Paint3D`). The cyclical behavior is aborted when the molecule is killed (by generating `killSignal`). The constructor of the class `Atom` is:

```
public Atom (Molecule molecule,String base,
             double x,double y,double z,
             double sx,double sy,double sz)
  {
     super (base,x,y,z,sx,sy,sz);
     this.molecule = molecule;
     killSignal = molecule.killSignal;
     this.behavior =
        SC.until (killSignal,
          SC.loop (
            SC.seq (
              SC.callback (constraintSignal,
                           new CollectConstraints (this)),
              SC.action (new Compute (this)),
              SC.action (new Paint3D (this)))));
  }
```

► The loop in definition of `behavior` is not instantaneous. This results from the presence of a callback in it, as callback instructions never terminate instantly. ◄

`CollectConstraints` is an action which simply adds to the atom the force sent to it:

```
public class CollectConstraints implements JavaCallback
{
    final Icobj3D me;
    public void execute (final ReactiveEngine _, final Object args)
        {
            Value3 v = (Value3)args;
            me.fx += v.x;
            me.fy += v.y;
            me.fz += v.z;
        }
    public CollectConstraints3D (Icobj3D me)
        {
            this.me = me;
        }
}
```

Class `Paint3D` is, for simplicity, not described here.

At construction, `Compute` creates a new instance of class `VerletV` to use the *Velocity-Verlet* resolution method, and a new instance of class `Newton` to hold the atom state. Each time it is executed, the `Compute` action applies one step of the resolution method and changes the atom state:

```
public class Compute implements JavaAction, Constants
{
    final Atom atom;
    final Newton newton;
    Method method;
    public void execute (final ReactiveEngine _)
        {
            method.step ();
            // set speed
            atom.sx = newton.state[1];
            atom.sy = newton.state[3];
            atom.sz = newton.state[5];
            // set coordinates
            atom.x = newton.state[0];
```

```
            atom.y = newton.state[2];
            atom.z = newton.state[4];
        }
    public Compute (Atom atom)
        {
            this.atom = atom;
            this.newton = new Newton (atom);
            method = new VerletV (newton,atom);
        }
}
```

4.2 Carbon Atom

A specific atom is a generic atom to which one adds the processing of LJ potentials, introduced in Section 5. One considers the case of a carbon atom. The mass, color, and radius of a carbon atom are defined in interface `Constants`. Basically, class `C` extends `Atom` by adding in parallel a new behavior which cyclically generates the event `CSignal` and process the three events corresponging to atoms `C`, `H`, and `O`. A value is associated to the generation of `CSignal`: it is the atom itself, which is returned by an object of the class `Self` described bellow. Definition of class `C` is:

```
public C (Molecule mol,double x,double y,double z,
                       double sx,double sy,double sz)
{
    super (mol,"C",x,y,z,sx,sy,sz);
    color = BLACK;
    mass = massC;
    radius = radiusC;

    behavior =
      SC.until (killSignal,
        SC.merge (
          behavior,
          SC.loop (
            SC.seq (
              SC.generate (CSignal,new Self (this)),
              SC.merge (
                  collect (CSignal,new LJPotential (ljC_C)),
                  collect (HSignal,new LJPotential (ljC_H)),
                  collect (OSignal,new LJPotential (ljC_O)))))));

    mol.addAtom (this);
}
```

► Assignment to field `behavior`, in class `C`, is by no mean a recursive definition. `behavior` appears both at the right and at the left of the sign "=", but in Java this sign does not denotes equality, but assignment. Actually, the statement should be reads as "evaluate the right part, and assign the result to the left part, which is `behavior`". Thus, the occurence of `behavior` appearing at right denotes the standard behavior of `Atom`, and the behavior of a carbon extends it by adding in parallel (`merge`) a specific behavior consisting in signaling itself and reacting to other atoms, according to their nature. ◄

`Self` defines a `JavaObjectExpression` always returning the object assigned to it at construction:

```
public class Self implements JavaObjectExpression
{
```

```
    Object self;
    public Object evaluateObject (final ReactiveEngine _)
        {
            return self;
        }
     public Self (Object self)
        {
            this.self = self;
        }
}
```

Parameters of the potentials corresponding to the various atoms are hold in variables `ljC_C`, `ljC_H`, and `ljC_O` defined in `Constants`. The call `collect(s,p)` collects all the values sent with event `s`, and applies the potential `p` to them (that is, calls the `execute` method of `CollectInteractions`, with p as parameter).

```
Program collect (Identifier signal,Potential potential)
{
   return SC.callback (signal,
      new CollectInteractions (potential,this));
}
```

The definition of `CollectInteractions` is:

```
public class CollectInteractions implements JavaCallback,Constants
{
    final Atom me;
    final Potential potential;
    public void execute (final ReactiveEngine _, final Object args)
        {
            Atom other = (Atom)args;
            if ( me == other ) return;
            Value3 f = potential.computeForce (other,me);
            me.fx += f.x;
            me.fy += f.y;
            me.fz += f.z;
       }
    public CollectInteractions (Potential potential,Atom me)
        {
            this.me = me;
            this.potential = potential;
        }
}
```

Two points are to be noticed: first, the equality test forbids the processing of the atom, named `me`, by itself; second, only `me` is transformed, not the atoms sent as values of events.

5 Lennard-Jones Potentials

Interface `Potential` specifies Lennard-Jones potentials, modeling van der Waals interactions between atoms. It defines a unique method `computeForce` which computes the vector representing the force issued from the potential existing between two atoms.

```
public interface Potential extends Constants
```

```
{
   public abstract Value3 computeForce (Atom source,Atom target);
}
```

A Lennard-Jones potential is defined by two parameters σ and ϵ. σ is the distance to which the potential is null, and ϵ is the depth of the potential at distance σ. The potential energy $\mathcal{U}(r_{ij})$ between two atoms i and j placed at distance r_{ij} is defined by:

$$\mathcal{U}(r_{ij}) = 4\epsilon((\frac{\sigma}{r_{ij}})^{12} - (\frac{\sigma}{r_{ij}})^{6}) \tag{3}$$

Class `LJPotential` implements `Potential` with method `computeForce` defined as:

```
public Value3 computeForce (Atom source,Atom target)
{
   energy = 0;
   if ( source.isLinked (target,maxDistanceAA) ) return zero;
   double r = Utils.distance (target,source);
   if ( r == 0 ) return zero;
   double sigma_on_r = sigma / r;
   double sigma_on_r_pow6 = Math.pow (sigma_on_r,6);
   double sigma_on_r_pow12 = sigma_on_r_pow6 * sigma_on_r_pow6;
   energy =  4 * eps * (sigma_on_r_pow12 - sigma_on_r_pow6);
   double flj =
     -24.0 * eps / r * (2*sigma_on_r_pow12 - sigma_on_r_pow6);
   Utils.setVector3d (v12,source,target);
   v12.normalize ();
   Utils.setVector3d (f,Utils.extProd (-flj,v12));
   res.x = f.x; res.y = f.y; res.z = f.z;
   return res;
}
```

Vector `zero` is returned if atoms `source` and `target` belong to the same molecule and are linked by at most `maxDistanceAA` bonds (`isLinked`). Otherwise, the distance `r` between the two atoms is computed, using the method `distance` defined in `Utils`. Vector `zero` is returned if `r` is 0, actually meaning that `source` and `target` are the same. Otherwise, energy is computed according to equation 3. Then, the module of the force (`flj`) is computed by deriving the energy. The result is the vector $\lambda\vec{v}$, where $\lambda = flj$ and $\vec{v}$ is the vector from `source` to `target`.

Energies of the various Lennard-Jones potentials are shown on Fig. 2.

6 Bonds

A bond is basically implemented by the abstract class `Spring` which extends `Icobj3D`. In this class, vector `f` holds the force applied to the two atoms `a` and `b` by means of two internal classes `GenA` and `GenB`. The module of the force is computed by the abstract method `controlLength` which is defined by concrete classes extending `Spring`. The behavior of `Spring` cyclically executes `controlLength` at each instant, and generates a constraint for `a` and `b`. The definition of `Spring` is:

```
public abstract class Spring extends Icobj3D
{
   final Atom a,b;
   Vector3d vab = new Vector3d ();
   Vector3d f = new Vector3d ();
```

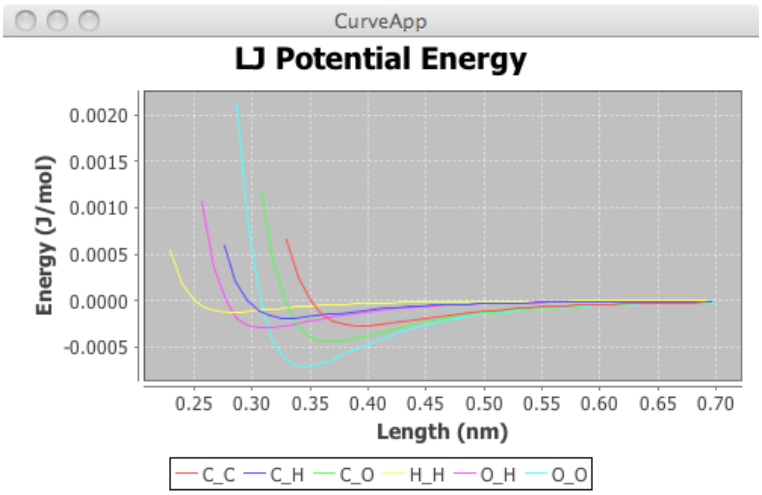


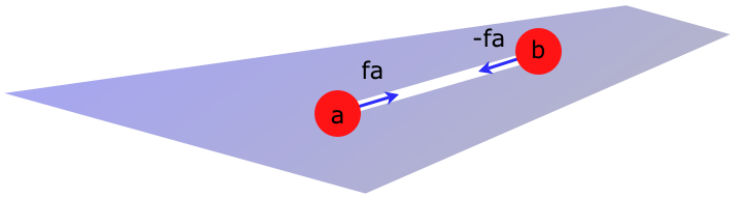
Figure 2: Lennard-Jones potentials

```
double energy;
public Spring (Molecule mol,Atom a,Atom b)
    {
        super ("s",0,0,0);
        this.a = a;
        this.b = b;

        this.behavior =
          SC.until (mol.killSignal,
            SC.loop (
              SC.seq (
                SC.action (new ControlLength ()),
                SC.generate (a.constraintSignal,new GenA ()),
                SC.generate (b.constraintSignal,new GenB ()),
                SC.stop () )));
    }
 public void applyForce (double force,Atom a,Atom b)
    {
        Utils.setVector3d (vab,a,b);
        vab.normalize ();
        Utils.extProd (f,force,vab);
    }
 abstract void controlLength ();
 public class ControlLength implements JavaAction
 {
    public void execute (final ReactiveEngine engine)
        {
            controlLength ();
        }
 }
 public class GenA implements JavaObjectExpression
 {
    Value3 res = new Value3 ();
    public Object evaluateObject (final ReactiveEngine engine)
        {
            res.set (f.x,f.y,f.z);
            return res;
        }
 }
```

```
    public class GenB implements JavaObjectExpression
    {                                                                    46
        Value3 res = new Value3 ();
        public Object evaluateObject (final ReactiveEngine engine)      48
            {
                res.set (-f.x,-f.y,-f.z);                                50
                return res;
            }                                                            52
    }
}                                                                        54
```

Method `applyForce` assigns to `f` a vector of the form $\lambda\vec{v}$, where $\vec{v}$ is the (normalised) vector betwen `a` and `b` and λ is obtained by deriving the potential. Note that the forces exerced on the two atoms are opposed, as shown on the following figure:



► The `stop` appearing in the loop of `behavior` definition prevents it to be instantaneous. This is one of the very few places where `stop` is mandatory. ◄

6.1 Harmonic Bond Potential

A harmonic bond potential defines the energy between two atoms i and j, with distance r_{ij} between them, as:

$$\mathcal{U}(r_{ij}) = k(r_{ij} - r_0)^2 \tag{4}$$

where k is the bond strength and r_0 is the constaint (equilibrium distance). The force associated to a harmonic potential has form $2k(r_{ij} - r_0)$. The class `HarmonicSpring` extends `Spring` by implementing method `controlLength`.

```
public class HarmonicSpring extends Spring
{                                                                        2
   double strength, length;
   public HarmonicSpring (Molecule mol,Atom a,Atom b,                    4
                          double strength,double length)
        {                                                                6
            super (mol,a,b);
            this.strength = strength;                                    8
            this.length = length;
        }                                                                10
   public void controlLength  ()
        {                                                                12
            double dist = Utils.distance (b,a);
            double diff = dist - length;                                 14
            energy = strength * diff * diff;
            double force = 2.0 * strength * diff;                        16
            applyForce (force,a,b);
        }                                                                18
}
```

► The use of an intermediary abstract class is justified because the modeling of scales larger than AA may need definitions of non-harmonic springs. These different springs could however still extend class `Spring`. ◄

Energies of the various bond potentials are shown on Fig. 3.

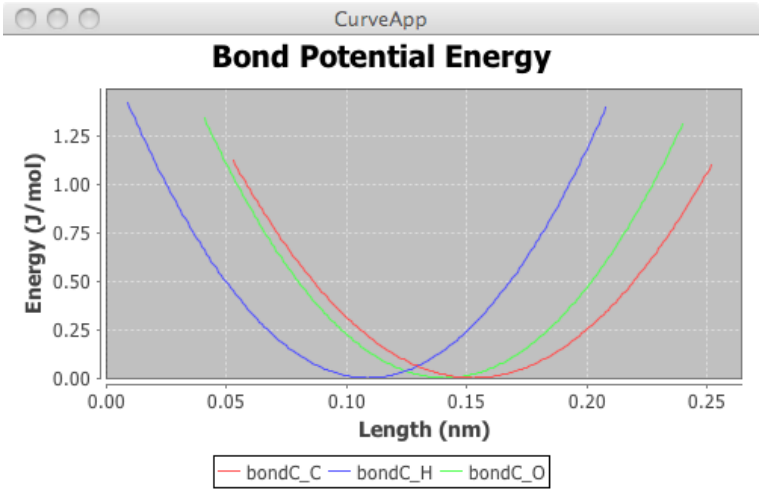


Figure 3: Bond potentials

7 Valence Angles

A valence angle is implemented by the abstract class `Angle`. The angle formed by three atoms is computed by method `valenceAngle` of `Angle` which returns a value in the interval $0, 2\pi$ and is defined as follows:

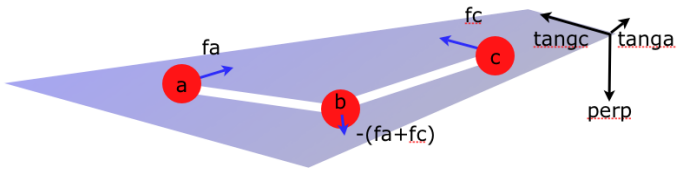
```
Vector3d ba = new Vector3d ();                                          1
Vector3d bc = new Vector3d ();
Vector3d perp = new Vector3d ();                                        3
Vector3d perp2 = new Vector3d ();
                                                                        5
public final double valenceAngle (Atom a,Atom b,Atom c)
    {                                                                   7
        Utils.setVector3d (ba,b,a);
        Utils.setVector3d (bc,b,c);                                     9
        double angle = ba.angle (bc);
        Utils.perp (perp,bc,ba);                                        11
        Utils.perp (perp2,perp,bc);
        // are perp2 and ba in the same direction ?                     13
        double dot = ba.dot (perp2);
       // angle in [0,2pi]                                              15
        if ( dot < 0 ) angle = angle + Math.PI;
        return angle;                                                   17
    }
```

Method `applyForce` of `Angle` applies a force on the three atoms:

```
    Vector3d tanga = new Vector3d ();
    Vector3d tangc = new Vector3d ();
    Vector3d sum = new Vector3d ();
    Vector3d inv = new Vector3d ();

    public void applyForce (double force,Atom a,Atom b,Atom c)
        {
            Utils.setVector3d (ba,b,a);
            Utils.setVector3d (bc,b,c);
            // perpendicular to plane p defined by ba and bc
            Utils.perp (perp,ba,bc);
            // perpendicular to -ba and perp in p
            Utils.perp (tanga,perp,Utils.opposite (inv,ba));
            tanga.normalize ();
            // perpendicular to bc and perp in p
            Utils.perp (tangc,perp,bc);
            tangc.normalize ();
            // forces on atoms
            Utils.extProd (fa,force,tanga);
            Utils.extProd (fc,force,tangc);
            Utils.opposite (fb,Utils.sum (sum,fa,fc));
        }
```

A drawing illustrates the application of the force:



7.1 Harmonic Valence Potential

A harmonic valence potential defines the energy between three atoms i, j, and k, forming an angle θ, as:

$$\mathcal{U}(\theta) = k(\theta - \theta_0)^2 \tag{5}$$

where k is the angle strength and θ_0 is the constaint (equilibrium angle). The force associated to an harmonic angle potential is $2k(\theta - \theta_0)$.

Class `HarmonicAngle` implements the abstract class `Angle` by defining the `controlAngle` method as:

```
void controlAngle ()
   {
      double angle = valenceAngle (a,b,c);
      // difference betweeen angle and consign
      double diff = angle - consign;
      // compute energy:  U(angle) = k(angle - consign)**2
      energy = strength * diff * diff;
      // compute force: - d/dt energy
      double force =  - 2.0 * strength * diff;
      // apply force
      applyForce (force,a,b,c)
   }
```

Energies of the various valence angle potentials of the force field used by the system are shown on Fig. 4.

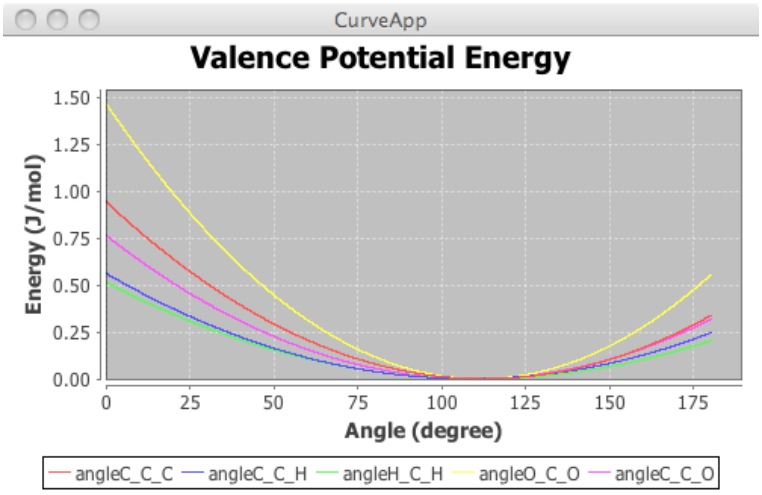


Figure 4: Valence potentials

8 Dihedrals

Dihedral potentials have the "triple cosine" form. The dihedral potential $\mathcal{U}(\theta)$ corresponding to a dihedral (torsion angle) θ is defined by:

$$\mathcal{U}(\theta) = 1/2((A_1(1 + cos(\theta + F_1)) + A_2(1 - cos(2\theta + F_2)) + A_3(1 + cos(3\theta + F_3)) + A_4) \quad (6)$$

The force is thus defined by

$$d\mathcal{U}(\theta)/dt = 1/2(A_1 sin(\theta + F_1) - 2A_2 sin(2\theta + F_2) + 3A_3 sin(3\theta + F_3)) \quad (7)$$

► Actually, we always consider the simpler form of dihedral potential, where A_4 and the F_i are null, but the system accepts the general triple-cosine form. ◄

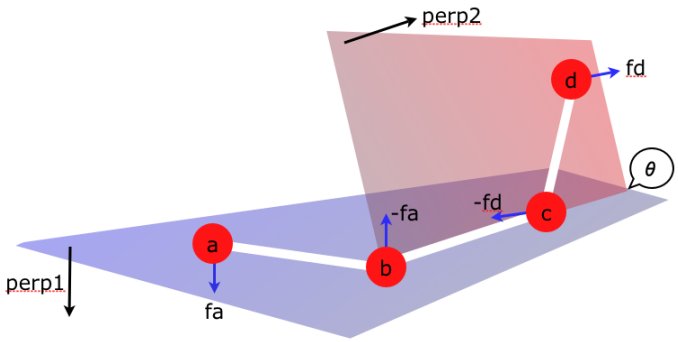
Class `Dihedral` defines dihedrals. The control of the torsion angle is performed by action `ControlAngle` defined as follows:

```
    Vector3d ba = new Vector3d ();
    Vector3d cd = new Vector3d ();                                          2
    Vector3d bc = new Vector3d ();
    Vector3d cb = new Vector3d ();                                          4

    Vector3d perp1 = new Vector3d ();                                       6
    Vector3d perp2 = new Vector3d ();
                                                                           8
    class ControlAngle implements JavaAction
    {                                                                      10
        public void execute (final ReactiveEngine _)
```

```
    {
        Utils.setVector3d (bc,b,c);
        Utils.setVector3d (cb,c,b);
        Utils.setVector3d (ba,b,a);
        Utils.setVector3d (cd,c,d);
        // vector perpendicular to plane1: a1,p1,p2
        Utils.perp (perp1,bc,ba);
        perp1.normalize ();
       // vector perpendicular to plane2: a2,p1,p2
        Utils.perp (perp2,cb,cd);
        perp2.normalize ();
        // measure dihedral angle [0,pi]
        double angle = perp1.angle (perp2);
        // are perp1 and cd in the same direction ?
        double dot = cd.dot (perp1);
       // angle in [-pi,+pi]
        if ( dot < 0 ) angle = -angle;
       // Math.PI is the consign
        angle = Math.PI + angle;
        // compute force
        double force = - computeForce (angle);
        // forces on atoms
        Utils.extProd (fa,force,perp1);
        Utils.extProd (fd,force,perp2);
        Utils.opposite (fb,fa);
        Utils.opposite (fc,fd);
    }
}
```

The control of the dihedral angle can be described by the drawing:



Method `computeForce` implements equations 6 and 7:

```
double computeForce (double theta)
    {
        // energy
        energy = 0.5 * (
            A1 * (1 + Math.cos (theta + F1)) +
            A2 * (1 - Math.cos (2*theta + F2)) +
            A3 * (1 + Math.cos (3*theta + F3))
            ) + A4;
            Statistics.incremDihedralEnergy (energy);
        // force
        double force = - 0.5 * (
            A1 * Math.sin (theta + F1) -
            (2 * A2 * Math.sin (2*theta + F2)) +
```

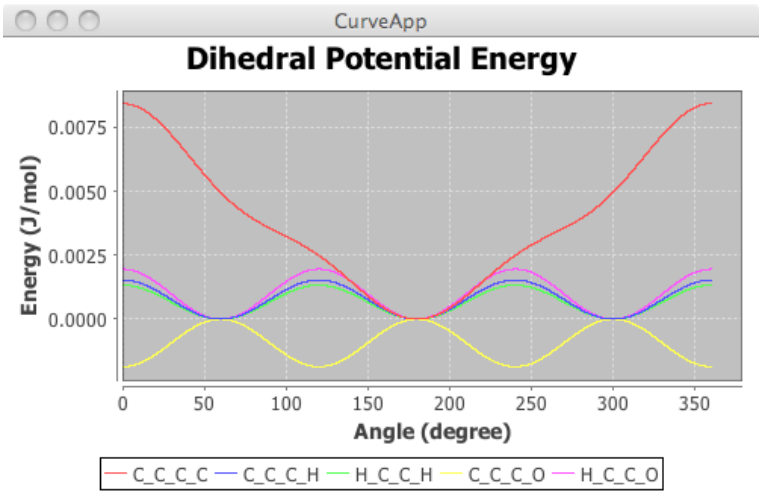


Figure 5: Dihedral potentials

```
                (3 * A3 * Math.sin (3*theta + F3))
                );                                                              15
            return force;
        }                                                                       17
```

Energies of the various dihedral potentials used in the system are shown on Fig. 5.

9 Molecules

A molecule of class `Molecule` is a container for its components which are stored in several vector data structures. For example, atoms are stored in `atomContent` and added in the molecule by method `addAtom` (see its use in Section 4.2). The `context` field contains the resolution time-step associated with the molecule (and also the resolution method and scale; we do not consider these points in more detail). The `build` method is the entry point to build the molecule; it should be redefined for specific molecules extending class `Molecule`. The molecule is registered in the simulation using method `registerIn`. The class `Molecule` has the form:

```
public class Molecule implements Constants                                      1
{
    String name,baseName,description;                                           3
    // initial coordinates
    double x,y,z;                                                               5
    // initial speed
    double sx,sy,sz;                                                            7
    // MD context
    MDContext context;                                                          9
    // Reactive workspace
    Simulation workspace;                                                       11
    // local scene for the molecule
    BranchGroup localScene;                                                     13
    // content of the molecule
    final Vector<Atom> atomContent = new Vector<Atom> ();                       15
    final Vector<Bond> bondContent = new Vector<Bond> ();
```

```
    final Vector<Angle> angleContent = new Vector<Angle> ();
    final Vector<Dihedral> dihedralContent = new Vector<Dihedral> ();
    // signals
    final Identifier killSignal = SC.simpleID ();
    public Molecule (String base,double x,double y,double z,
                                                double sx,double sy,double sz)
        {
            this (base,x,y,z,sx,sy,sz,new MDContext (0,VERLETV,Scale.AA));
        }
    public void addAtom (Atom atom)
        {
            atomContent.add (atom);
            atom.injectIn (localScene);
        }
    public void build () { }
    public void registerIn (Simulation workspace)
        {
            for ( Enumeration e = atomContent.elements (); e.hasMoreElements (); ) {
                Atom atom = (Atom)e.nextElement ();
                atom.registerIn (workspace);
            }
            ....
            workspace.register (this);
            workspace.scene.addChild (localScene);
        }
     ....
}
```

9.1 Carbon Chains

A carbon chain is basically a molecule made of a chain of carbon atoms to which other atoms are linked. In the rest of this section, we describe the building of carbon chains of the form CH3-(CH2)*n-CH3. The goal of the construction is to build molecules with minimum of potential energy. Examples of carbon chains are given in section 10.

The number of carbon atoms is a parameter of the construction, coded in `cNum`. Carbon atoms are placed in the array `backbone` and the other atoms (hydrogen and oxigen atoms) are placed in the array `others`. Class `CarbonChain` is the basic class to model carbon chains:

```
public class CarbonChain extends Molecule
{
    int cNum;
    Atom backbone [];
    Atom[] others [];

    double lCH = bondC_H[1];
    double aHCH = angleH_C_H[1];
    double aCCH = angleC_C_H[1];

    public CarbonChain (Simulation workspace,int cNum,
                        double x,double y,double z,
                        double sx,double sy,double sz)
        {
            super ("carbonChain",x,y,z,sx,sy,sz);
            this.workspace = workspace;
            this.cNum = cNum;
            this.backbone = new Atom [cNum];
            this.others = new Atom[cNum] [];
            for ( int k = 0; k < cNum; k++ ) others[k] = new Atom [0];
        }
```

```
    ....
}
```

The `build` method of `CarbonChain` consists in a sequence of steps: `buildBackbone` fills the array `backbone`; three hydogen atoms are added by method `addTop` to the first carbon; two hydogens atoms are added to each carbon, except the first and the last, by method `addH2`; three hydrogens are added to the last carbon by `addBottom`; finally, bonds, valence angles, and dihedrals are added, using `createBonds`, `createAngles`, and `createDihedrals`; despite the fact that the same technique is used for the 3 methods, we give their complete definition, for the sake of completeness. Definition of `build` is:

```
public void build ()
  {
     buildBackbone ();
     addTop ();
     for ( int k = 1; k < cNum-1; k++ ) addH2 (k);
     addBottom ();
     createBonds ();
     createAngles ();
     createDihedrals ();
  }
```

Filling the backbone is performed by:

```
double incrX = bondC_C[1] * Math.cos (angleC_C_C [1] / 2);
double incrY = bondC_C[1] * Math.sin (angleC_C_C [1] / 2);

void buildBackbone ()
  {
    double currentY = y;
    double currentX = x;
    for ( int k = 0; k < cNum; k++ ) {
      if ( k%2 == 0 ) currentX -=incrX; else currentX += incrX;
      backbone[k] = new C (this,currentX,currentY,z,sx,sy,sz);
      currentY -= incrY;
    }
  }
```

9.2 Hydrogen Atoms

The three methods `addTop`, `addH2`, and `addBottom` are used to place hydrogen atoms. As `addTop` and `addBottom` are very similar, we only describe the last one (it is redefined in Section 10.2). The `addH2` method attaches two hydogen atoms to a carbon atom (CH2):

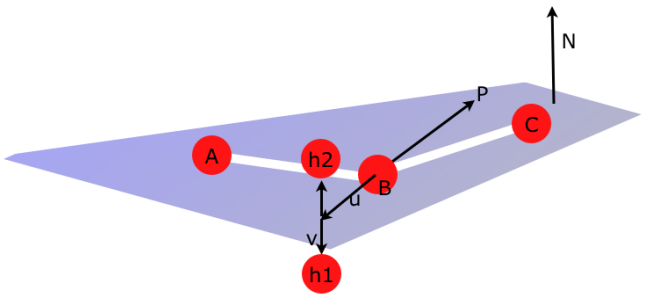
```
double cos = lCH * Math.cos (aHCH /2);
double sin = lCH * Math.sin (aHCH /2);

void addH2 (int k)
  {
    Atom A = backbone[k-1];
    Atom B = backbone[k];
    Atom C = backbone[k+1];
    Vector3d BA = Utils.vect (B,A);
```

```
    Vector3d BC = Utils.vect (B,C);
    Vector3d P = Utils.normalize (Utils.sum (BA,BC));
    Vector3d N = Utils.normalize (Utils.perp (BA,BC));

    Vector3d u = Utils.extProd (-cos,P);
    Vector3d v = Utils.extProd (-sin,N);
    Vector3d w = Utils.sum (u,v);
    Vector3d q = Utils.sum (u,Utils.opposite (v));

    Atom h1 = new H (this,B.x+w.x, B.y+w.y, B.z+w.z, B.sx,B.sy,B.sz);
    Atom h2 = new H (this,B.x+q.x, B.y+q.y, B.z+q.z, B.sx,B.sy,B.sz);
    others [k] = new Atom [2];
    others [k][0] = h1;
    others [k][1] = h2;
  }
```



► For vector manipulations, we prefer here to use direct methods (e.g. `Utils.vect`) instead of in-place methods (e.g. `Utils.setVector3d`). This is because there is no need for optimisation, as we are just building molecules, not executing them. ◄

The `addBottom` method places 3 hydrogen atoms attached to the last carbon atom (CH3):

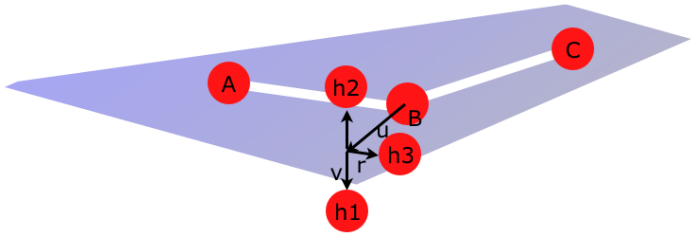
```
double cos2 = lCH * Math.cos (aCCH/2);
double sin2 = lCH * Math.sin (aCCH/2);

void addBottom ()
  {
    Atom A = backbone[cNum-3];
    Atom B = backbone[cNum-2];
    Atom C = backbone[cNum-1];
    Vector3d AB = Utils.vect (A,B);
    Vector3d P = Utils.normalize (Utils.opposite (Utils.sum (AB,Utils.vect (C,B))));
    Vector3d N = Utils.normalize (Utils.perp (AB,P));

    Vector3d u = Utils.extProd (cos,P);
    Vector3d v = Utils.extProd (sin,N);
    Vector3d w = Utils.sum (u,v);
    Vector3d q = Utils.sum (u,Utils.opposite (v));

    Atom h1 = new H (this,C.x+w.x, C.y+w.y, C.z+w.z, C.sx,C.sy,C.sz);
    Atom h2 = new H (this,C.x+q.x, C.y+q.y, C.z+q.z, C.sx,C.sy,C.sz);
    Vector3d perp = Utils.normalize (Utils.perp (u,v));
    Vector3d r =
      Utils.sum (Utils.extProd (-cos2,Utils.normalize (u)),
                 Utils.extProd (sin2,perp));
```

```
    Atom h3 = new H (this,C.x+r.x, C.y+r.y, C.z+r.z, C.sx,C.sy,C.sz);
    others [cNum-1] = new Atom [3];
    others [cNum-1][0] = h1;
    others [cNum-1][1] = h2;
    others [cNum-1][2] = h3;
}
```



9.3 Creation of Bonds

Harmonic bonds are created between backbone atoms, and between a carbon and the hydrogen or oxygen atoms on the same plane.

```
void createBonds ()
    {
        for ( int k = 0; k < cNum - 1; k++ ) {
            new HarmonicBond (this,backbone[k],backbone[k+1],bondC_C);
        }
        for ( int k = 0; k < cNum; k++ ) {
            Atom c = backbone [k];
            for ( int l = 0; l < others [k].length; l++ ) {
                Atom a = others[k][l];
                if ( a instanceof H ) new HarmonicBond (this,c,a,bondC_H);
                else if ( a instanceof O ) new HarmonicBond (this,c,a,bondC_O);
            }
        }
    }
```

9.4 Creation of Valence Angles

Creation of valence angles is done by `createAngles` method:

```
void createAngles ()
    {
        // backbone angles
        for ( int k = 0; k < cNum - 2; k++ ) {
            new HarmonicAngle (this,backbone[k],
                    backbone[k+1],backbone[k+2],angleC_C_C);
        }
        for ( int k = 0; k < cNum; k++ ) {
            Atom c = backbone [k];
            int len = others [k].length;
            for ( int l = 0; l < len; l++ ) {
                Atom a = others[k][l];
                for ( int m = l+1; m < len; m++ ) {
```

```
                    Atom b = others[k][m];
                    if ( a instanceof H )
                       new HarmonicAngle (this,a,c,b,angleH_C_H);
                    else if ( a instanceof O )
                       new HarmonicAngle (this,a,c,b,angleO_C_O);
                }
            }
        }
        for ( int k = 0; k < cNum - 1; k++ ) {
            Atom c1 = backbone [k];
            Atom c2 = backbone [k+1];
            int len = others [k].length;
            for ( int l = 0; l < len; l++ ) {
                Atom a = others[k][l];
                new HarmonicAngle (this,a,c1,c2,angleC_C_H);
            }
        }
        for ( int k = 1; k < cNum; k++ ) {
            Atom c1 = backbone [k-1];
            Atom c2 = backbone [k];
            int len = others [k].length;
            for ( int l = 0; l < len; l++ ) {
                Atom a = others[k][l];
                if ( a instanceof H )
                   new HarmonicAngle (this,c1,c2,a,angleC_C_H);
                else if ( a instanceof O )
                  new HarmonicAngle (this,c1,c2,a,angleC_C_O);
            }
        }
    }
```

9.5 Creation of Dihedrals

Creation of dihedrals is done by `createDihedrals` method:

```
  void createDihedrals ()
      {
          for ( int k = 0; k < cNum-3; k++ ) {
              new Dihedral (this,backbone[k],backbone[k+1],
                            backbone[k+2],backbone[k+3],dihedralC_C_C_C);
          }
          for ( int k = 0; k < cNum-2; k++ ) {
              Atom c1 = backbone [k];
              Atom c2 = backbone [k+1];
              Atom c3 = backbone [k+2];
              int len = others [k].length;
              for ( int l = 0; l < len; l++ ) {
                  Atom a = others[k][l];
                  new Dihedral (this,a,c1,c2,c3,dihedralC_C_C_H);
              }
          }
          for ( int k = 0; k < cNum-2; k++ ) {
              Atom c1 = backbone [k];
              Atom c2 = backbone [k+1];
              Atom c3 = backbone [k+2];
              int len = others [k+2].length;
              for ( int l = 0; l < len; l++ ) {
                  Atom a = others[k+2][l];
                  if ( a instanceof H )
                     new Dihedral (this,c1,c2,c3,a,dihedralC_C_C_H);
                  else if ( a instanceof O )
```

```
                    new Dihedral (this,c1,c2,c3,a,dihedralC_C_C_O);
                }
            }
            for ( int k = 0; k < cNum-1; k++ ) {
                Atom c1 = backbone [k];
                Atom c2 = backbone [k+1];
                for ( int l = 0; l < others [k].length; l++ ) {
                    Atom a = others[k][l];
                    for ( int m = 0; m < others [k+1].length; m++ ) {
                        Atom b = others[k+1][m];
                        if ( b instanceof H )
                            new Dihedral (this,a,c1,c2,b,dihedralH_C_C_H);
                        else if ( b instanceof O )
                            new Dihedral (this,a,c1,c2,b,dihedralH_C_C_O);
                    }
                }
            }
        }
```

10 Examples

In this section, we simulate alkane and acid molecules which are specific carbon chains. All the carbon chains considered in this section are made of 8 carbon atoms. In subsection 10.3, we put the focus on stability of the resolution method.

10.1 Alkane

A typical program extends `Simulation` (or simply `Workspace3D`) and defines a constructor in which the `createUniverse` method is called, to start Java3D. The `main` entry point of the program calls the constructor (possibly using the `standAlone` function, which produces a window at screen).

For example, here is a simulation of an alkane molecule with 8 carbon atoms. The molecule is made of 25 atoms, 24 bonds, 45 valence angles, and 60 dihedrals. The time-step used is 1 femto-second. A screenshot is shown on Fig. 6. The program is:

```
public class SimpleApp extends Simulation
{
    public SimpleApp ()
        {
            super ("SimpleApp");
            backgroundColor =CYAN;
            createUniverse ();
            CarbonChain mol = new CarbonChain (this,8,0,0.4,0, 0,0,0);
            mol.build ();
            mol.context.timeStep = 0.001; // 1 fs
            mol.registerIn (this);
        }
    public static void main (String [] args)
        {
            standAlone (new SimpleApp ());
        }
}
```

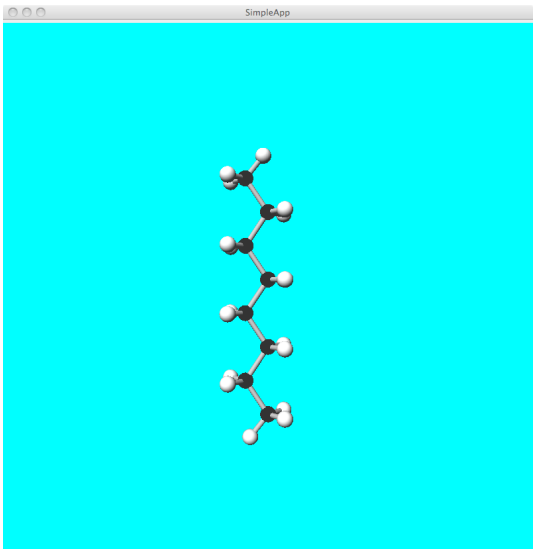


Figure 6: Alkane carbon chains

10.2 Acid

An `Acid` molecule is a carbon chain in which the bottom part is changed to CO2 instead of CH3. It thus extends `CarbonChain` by redefining method `addBottom`:

```
class Acid extends CarbonChain                                          1
{
   public Acid (Simulation workspace,int cNum,                          3
                double x,double y,double z,
                double sx,double sy,double sz)                          5
        {
            super (workspace,cNum,x,y,z,sx,sy,sz);                      7
            this.description = "CH3-(CH2)*"+(cNum-2)+"-CO2";
        }                                                               9
   void addBottom ()
        {                                                               11
            Atom A = backbone[cNum-3];
            Atom B = backbone[cNum-2];                                  13
            Atom C = backbone[cNum-1];
                                                                        15
            double lCO = bondC_O[1];
            double aOCO = angleO_C_O[1];                                17
            double cos = lCO * Math.cos (aOCO /2);
            double sin = lCO * Math.sin (aOCO /2);                      19

            Vector3d AB = Utils.vect (A,B);                             21
            Vector3d CB = Utils.vect (C,B);
            Vector3d P =                                                23
                Utils.normalize (
                   Utils.opposite (                                     25
                      Utils.sum (AB, Utils.vect (C,B))));
            Vector3d N = Utils.normalize (Utils.perp (AB,P));           27

            Vector3d u = Utils.extProd (cos,P);                         29
            Vector3d v = Utils.extProd (sin,N);
            Vector3d w = Utils.sum (u,v);                               31
```

```
            Vector3d q = Utils.sum(u, Utils.opposite(v));
                                                                                    33
            Atom o1 = new O(this, C.x+w.x, C.y+w.y, C.z+w.z, C.sx, C.sy, C.sz);
            Atom o2 = new O(this, C.x+q.x, C.y+q.y, C.z+q.z, C.sx, C.sy, C.sz);     35
            others[cNum-1] = new Atom[2];
            others[cNum-1][0] = o1;                                                 37
            others[cNum-1][1] = o2;
        }                                                                           39
}
```

Replacing in class `SimpleApp CarbonChain` by `Acid`, on obtains the simulation of an acid molecule. A screenshot is shown on Fig. 7.

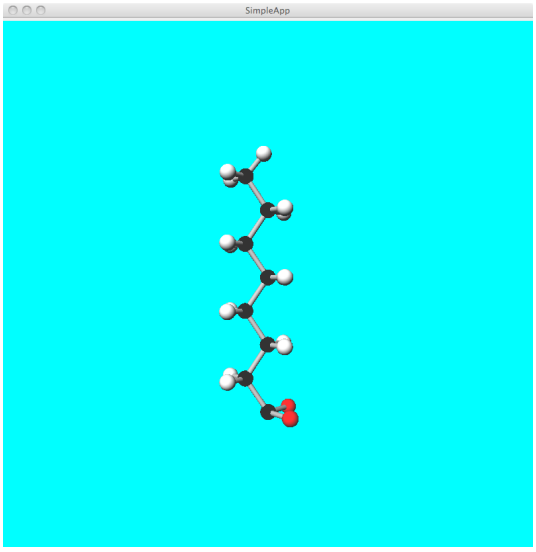


Figure 7: Acid carbon chain

10.3 Stability

In this section, one considers the stability of the resolution method, with the aid of several simulations. One must keep in mind that simulations do not contain means to control (kinetic) energy (for example, thermostat); dynamics of molecules only depends of the force field and initial conditions.

Simulation 1

We first simulate an alkane molecule during 300 ps with a resolution time-step of 1 fs (the simulation thus takes 600,000 instants). One traces the various sources of potential energy: bonds, valence angles, dihedrals, and LJ potential (samples are taken every ps). The result is shown on Fig. 8. One sees that energy is globally stable during the first 300 ps.

Energy start growing after 300 ps, with valence angles, and the increase propagates to dihedrals while bond and LJ energies stay stable. This is shown on Fig. 9.

Energy up to 700 ps is shown on Fig. 10. One sees that divergence reaches bonds and LJ energies near 0.5 nano-second.

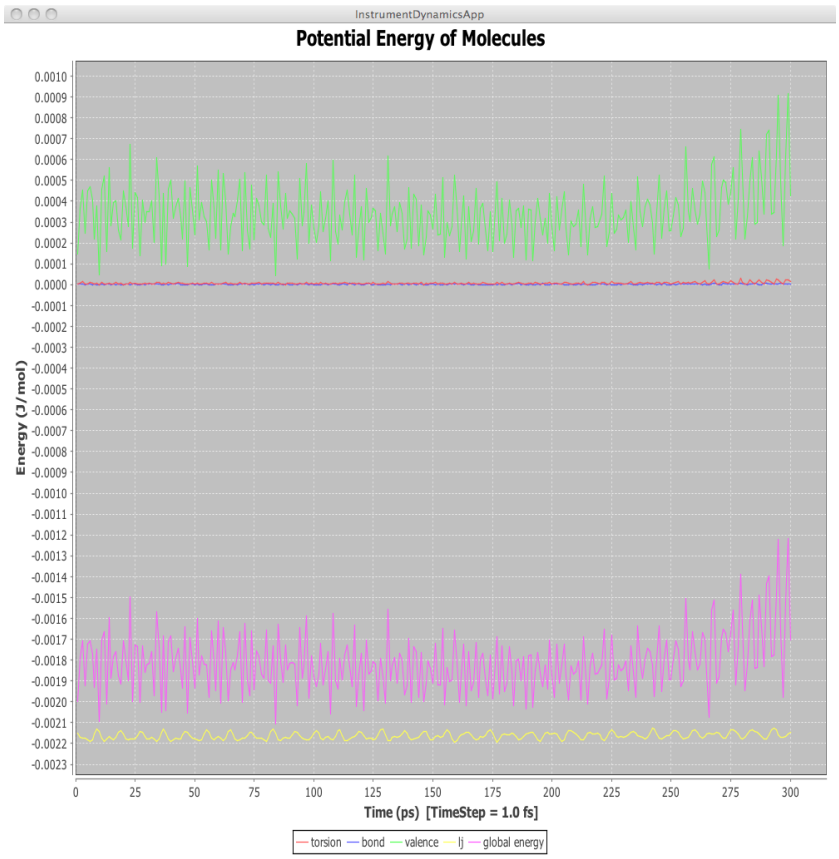


Figure 8: Alkane, 300 ps, time-step 1 fs

Simulation explodes just after 700 ps. Here is a trace of total energy at some instants around explosion (recall that one fs of simulation needs two instants, and that total energy is the sum of potential and kinetic energies):

```
instant 1405958: total energy: 12.329925681290062
instant 1405965: total energy: 1194.9563097546588
instant 1406016: total energy: 4285583.1626097085
instant 1406017: total energy: 1.7158277146908313E7
```

Simulation 2

We consider the same simulation, but with a time-step of 0.5 femto-second to see if a smaller time-step increases stability. The evolution of energy is shown on Fig. 11. One sees that, for this simulation, a smaller time-step does not significantly change the evolution of energy. However, explosion of the simulation is delayed.

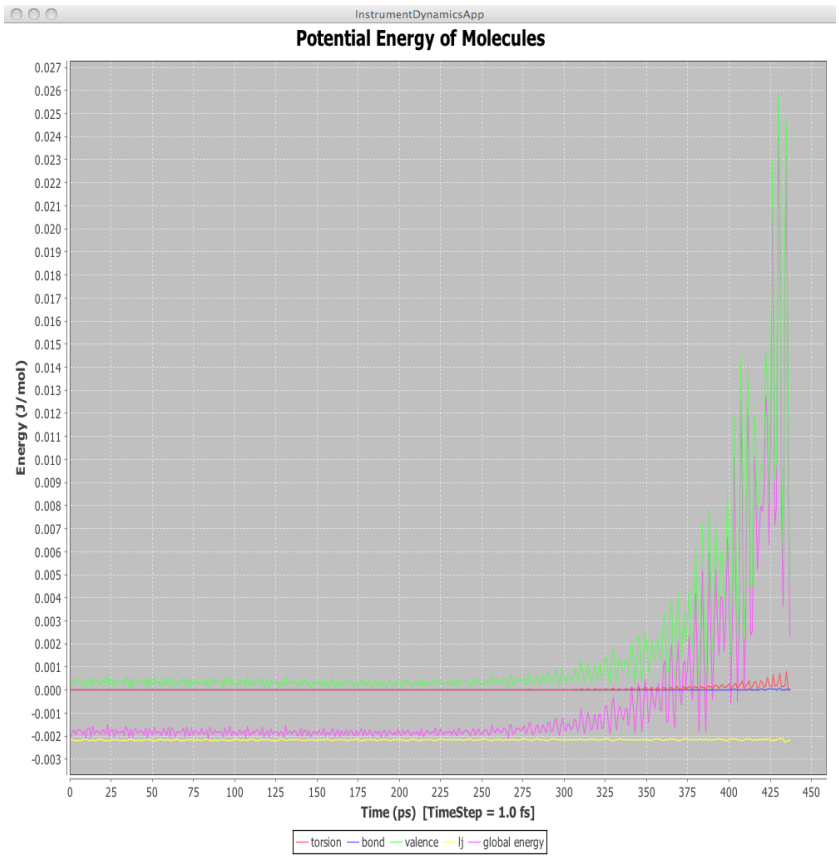


Figure 9: Alkane, 400 ps, time-step 1 fs

Simulation 3

We now turn to acid molecules and simulates one acid molecule during 300 ps, with a time-step of 1 fs. The result is shown on Fig. 12. Actually, the replacement of the molecule bottom makes the simulation less stable, as energy start to increase sooner .

Simulation 4

We now simulate a system made of three 8-acid molecules (time-step is 1 fs). The system after 300 ps is shown on Fig. 13 and the energy evolution is shown on Fig. 14. Energy evolution during 0.5 ns is shown on Fig. 15; divergence comes very soon after this time.

11 Conclusion

We have presented a MD system based on Java and using the RP framework SugarCubes for logical parallelism. Definion and implemention of atoms and molecules have been described in detail, and some results of simulations showing the stability of the resolution method were given.

The use of RP for MD simulations increases modularity: while object-orientation gives means

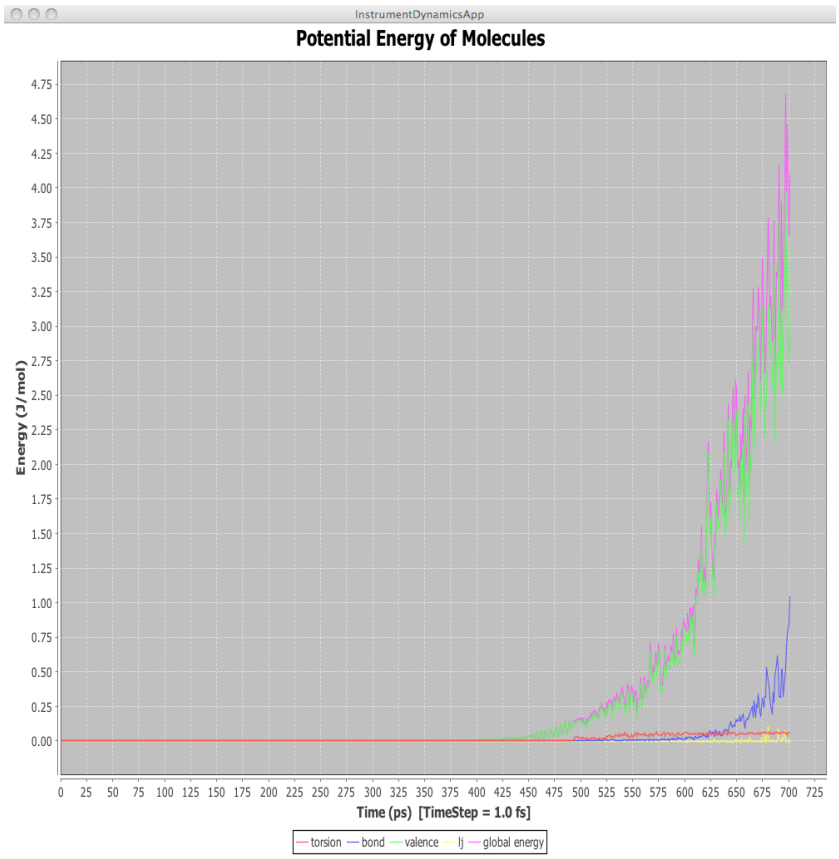


Figure 10: Alkane, 700 ps, time-step 1 fs

to re-use data, RP gives means (basically, logical parallelism) to re-use behaviors. RP is based on a discretisation of time (instants) which appears natural in the resolution mechanism of MD. Instantaneous interactions which are at the basis of classical physics can be naturally expressed in RP through instantaneously broadcast events. Finally, full determinism of the `merge` parallel operator of SugarCubes makes simulations totally deterministic. This corresponds to the deterministic nature of classical physics and is mandatory to obtain simulations reversible in time.

In a future work, we plan to extend our system to multi-scale, multi-time-step simulations. We also plan to study ways to benefit from real parallelism (issued from multicores, multiprocessors, or clusters of machines) to get more efficient simulations.

References

[1] DL_POLY. http://www.cse.scitech.ac.uk/ccg/software/DL_POLY.

[2] Java3D. http://www.java3d.org.

[3] Reactive Programming. http://www-sop.inria.fr/indes/rp.

[4] Multiscale Simulation Methods in Molecular Sciences. *NIC Series, Forschungszentrum Julich, Germany*, vol. 42, Winter School, 2-6 March, 2009.

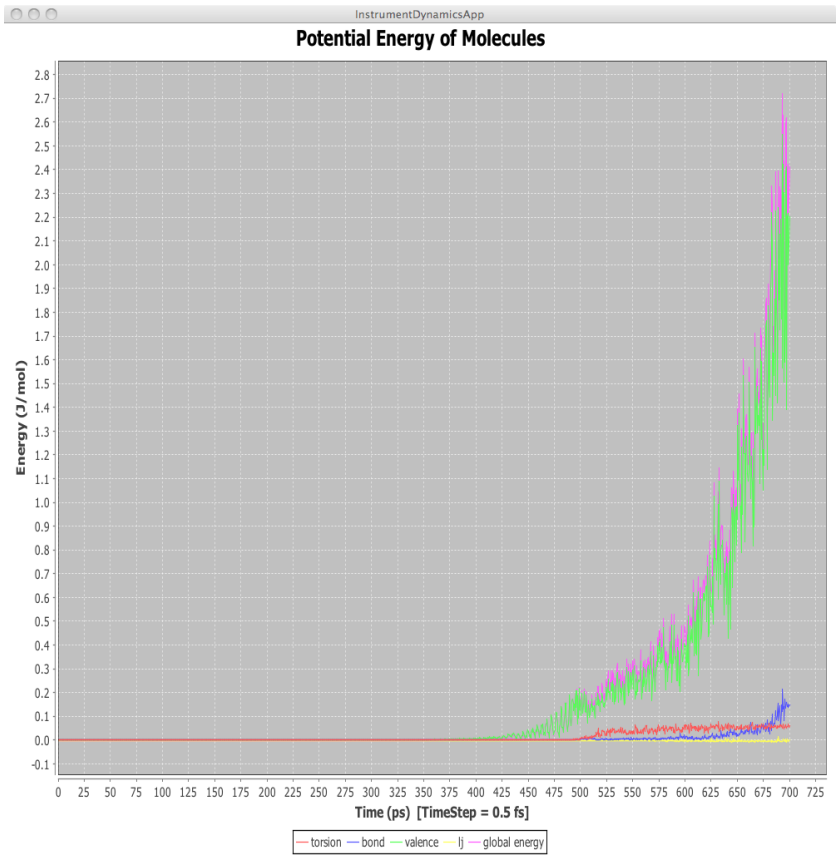


Figure 11: Alkane, 700 ps, time-step 0.5 fs

[5] M. P. Allen and D. J. Tildesley. *Computer Simulation of Liquids*. Oxford, 1987.

[6] K. Arnold and J. Goslin. *The Java Programming Language*. Addison-Wesley, 1996.

[7] F. Boussinot. Reactive C: An Extension of C to Program Reactive Systems. *Software Practice and Experience*, 21(4):401–428, april 1991.

[8] F. Boussinot and J-F. Susini. The SugarCubes Tool Box - A Reactive Java Framework. *Software Practice and Experience*, 28(14):1531–1550, december 1998.

[9] W. Damm, A. Frontera, J. Rirado-Rives, and W. L. Jorgensen. OPLS All-Atom Force Field for Carbohydrates. *Journal of Computational Chemistry*, 18(16):1955–1970, 1997.

[10] L. Mandel and M. Pouzet. ReactiveML, A Reactive Extension to ML. In *ACM International conference on Principles and Practice of Declarative Programming (PPDP'05)*, Lisbon, Portugal, July 2005.

[11] Loup Verlet. Computer "Experiments" on Classical Fluids. I. Thermodynamical Properties of Lennard-Jones Molecules. *Phys. Rev.*, 159:98–103, Jul 1967.

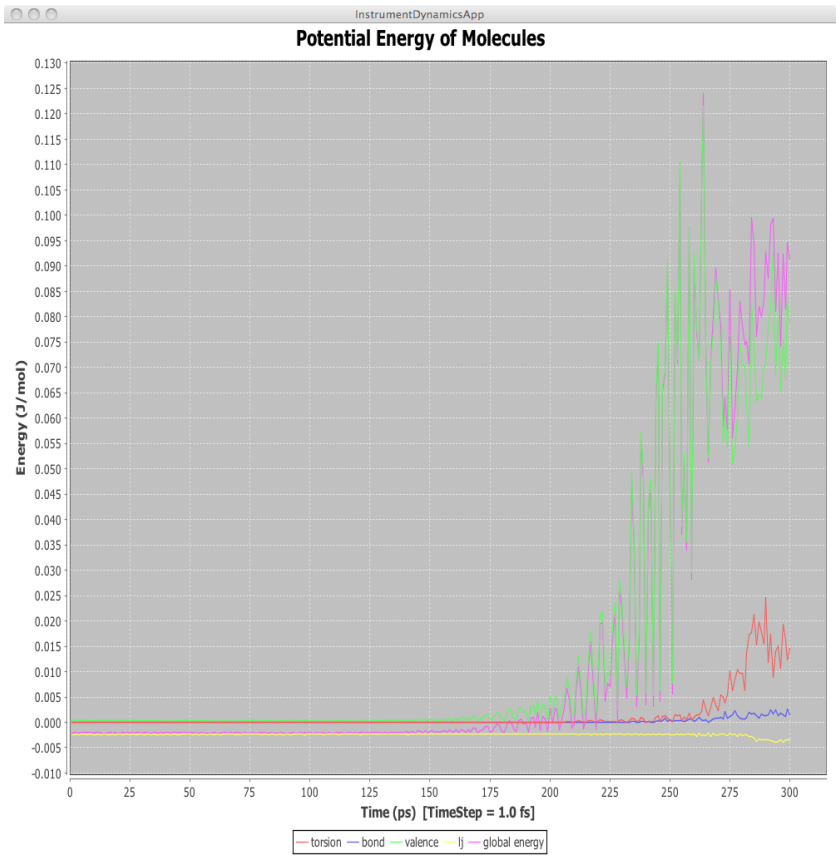


Figure 12: Acid, 300 ps, time-step 1 fs

Contents

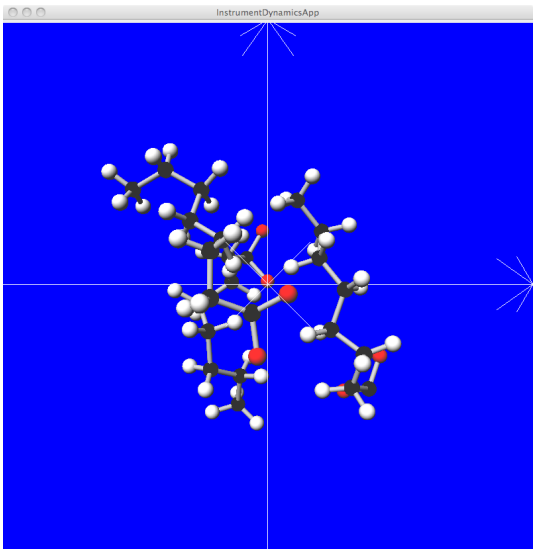


Figure 13: Result of a 300 ps simulation of 3 acid molecules

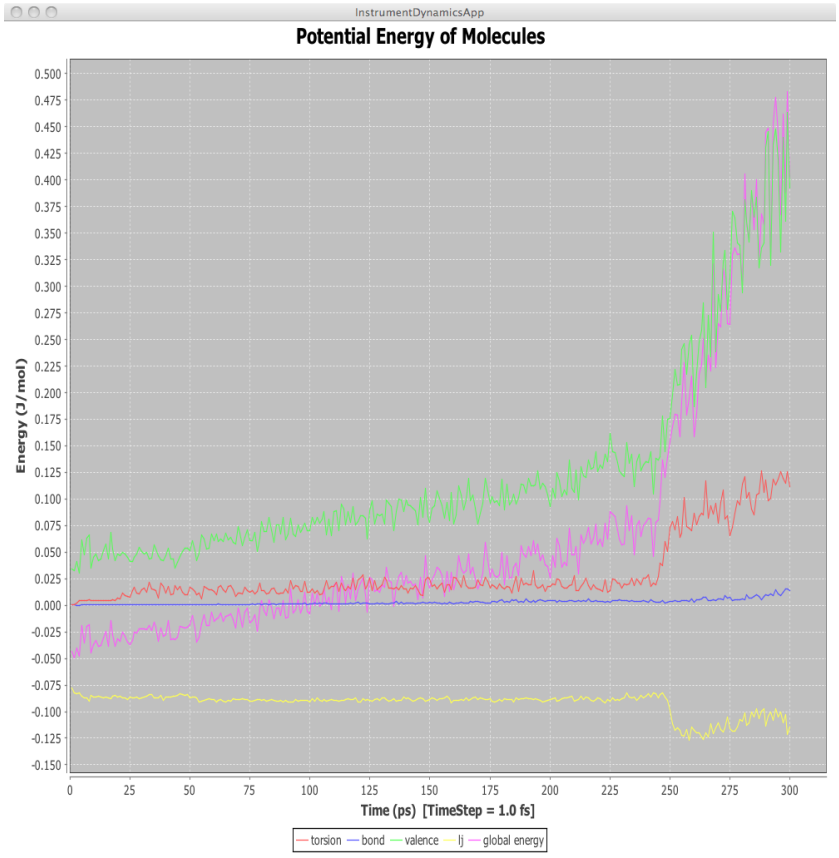


Figure 14: 3 acid molecules, 300 ps, time-step 1 fs

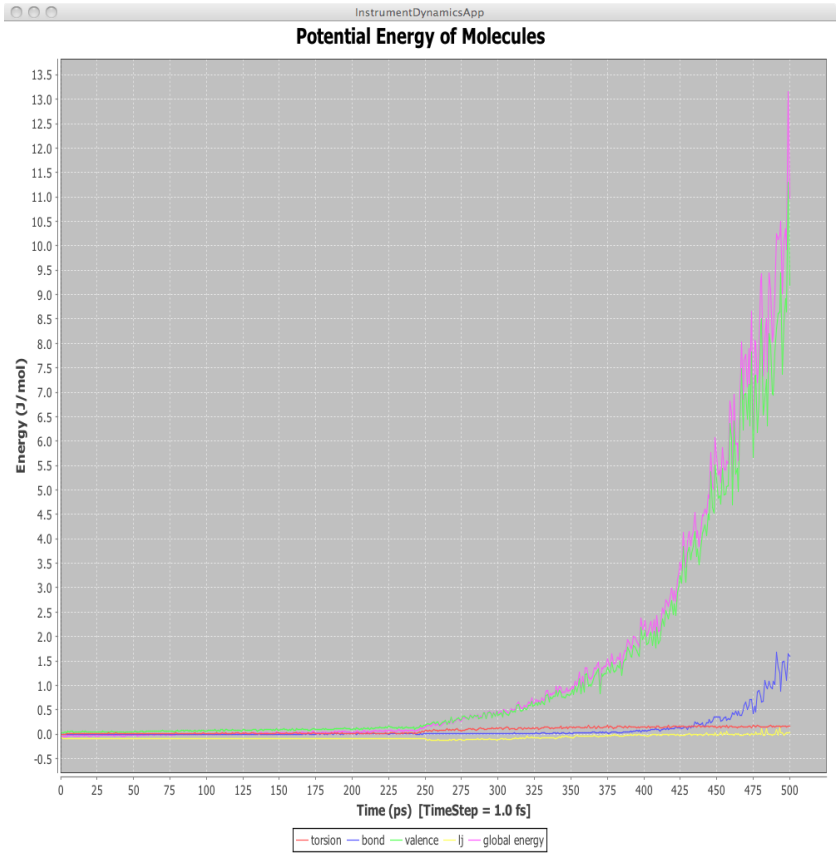


Figure 15: 3 acid molecules, 500 ps, time-step 1 fs